Lab 4: Multivariate Methods

CS 412

This lab can be conducted in groups or individually.

In this lab, we will delve into multi-variate Gaussian distribution, including its inference, sampling, and visualization. There are a lot of plotting in this lab, and they cannot be auto-graded.

Deadline: **23:59, Mar 3**.

Please refer to `Lab_Guideline.pdf` in the same Google Drive folder as this Jupyter notebook; the guidelines there apply to all the labs.

To start, let us import the necessary packages.

```
# set up code for this experiment
import numpy as np
import scipy.linalg
import matplotlib.pyplot as plt

%matplotlib inline

np.random.seed(1)
```

Problem 1: Exercise 4.1 of Alpaydin (4 pt)

Write the code that generates a Bernoulli sample with given parameter p, and the code that calculates $\hat{p}$ from the sample.

The sampler will be based on the function `np.random.random_sample`, which returns (an array of) uniformly distributed samples from [0.0, 1.0). To turn them into Bernoulli samples, just threshold with p. Then use the resulting binary value to compute the estimate $\hat{p}$. So we are done in three steps.

Important: for efficiency, you are **not** allowed to use loops in your

implementation, meaning that you should not call `np.random.random_sample` to just return a single random number.

```
def ex4_1(p, nSample):
  """
  Inputs:
  - p: a real number, which specifies the parameter of Bernoull
  - nSample: an integer which is the  number of samples to draw

  Output:
  - phat: the estimate of p from the samples
  """
  np.random.seed(1)
  # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

  temp = np.random.random_sample(size=nSample)
  phat = np.sum(temp < p) / nSample

  # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

  return phat

"""
Unit test
Should print: some value close to 0.2 (it's random)
"""
p = 0.2
nSample = 1000
phat = ex4_1(p, nSample)
print (phat)
```

```
0.203
```

Problem 2: Exercise 4.3 of Alpaydin (23 pt)

Write the code that generates a real-valued normal sample with given μ and σ, and the code that calculates m and s from the sample. Do the same using the Bayes' estimator assuming a Gaussian prior distribution for μ.

2.1 Implement the density function of a Gaussian distribution (4 pt)

Write a function that, given the value of mean μ and standard deviation σ, computes the density of the univariate Gaussian distribution at x. Here x can be a vector, and the result should be a vector of the same size, with the i-th element being the density of x[i].

```
def nrmf(x, mu, sigma):
  '''
  Given mean mu and standard deviation sigma,
  compute the density of the univariate Gaussian distribution a
  Here x can be a vector, and the result should be a vector of
  with the i-th element being the density of x[i].
  Input:
    - x:    a 1-D numpy array specifying where to query the pro
    - mu:   a real number specifying the mean of the Gaussian
    - sigma: a real number specifying the standard deviation of
  Outputs:
    - p:  a 1-D numpy array specifying the pdf at each element
  '''
  # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)***
  p = (1 / (np.sqrt(2 * np.pi) * sigma)) * np.exp(-np.square(x
  # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

  return p

'''
Unit test
We compare our result with the density computed by scipy
'''
import scipy.stats

x = np.random.random_sample(3)
mu = 1
sigma = 2
density = nrmf(x, mu, sigma)
ref_density = scipy.stats.norm(mu, sigma).pdf(x)
print(density)
print(ref_density)
```

```
[0.18845383 0.19916872 0.1979399 ]
[0.18845383 0.19916872 0.1979399 ]
```

2.2 Implement the Bayes' estimator assuming a Gaussian prior distribution for μ. (6 pt)

In a Gaussian distribution $N(\mu, \sigma^2)$, assume μ has a Gaussian prior N (priorMean, priorStd^2). Then compute μ_{Bayes}: the Bayes' estimate of mu by using the equation on slide 8 of Parametric Methods lecture slides: $E[\theta|X] = \ldots$.

```
def bayes_estimator(x, sigma, priorMean, priorStd):
  '''
    Compute mu_bayes: the Bayes' estimate of mu by using the ec
  Input:
    - x:    a 1-D numpy array specifying the samples from the G
    - sigma: a real number specifying the standard deviation of
    - priorMean: a real number specifying the mean of the Gauss
    - priorStd: a real number specifying the standard deviation
  Outputs:
    - mu_post: a real number specifying the Bayes' estimate of
  '''

  # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)***
  N = x.shape[0]
  m = np.mean(x)
  mu_post = ((N / sigma**2) / ((N/sigma**2) + (1/priorStd**2)))

  # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
  return mu_post

'''
Unit test
You should get 1.25
'''
np.random.seed(1)
sigma = 2
priorMean, priorStd = (-1, 2)
x = np.array([1, 2, 3])
bayes_estimator(x, sigma, priorMean, priorStd)
```

```
1.25
```

2.3 Do the following computation and plotting (8

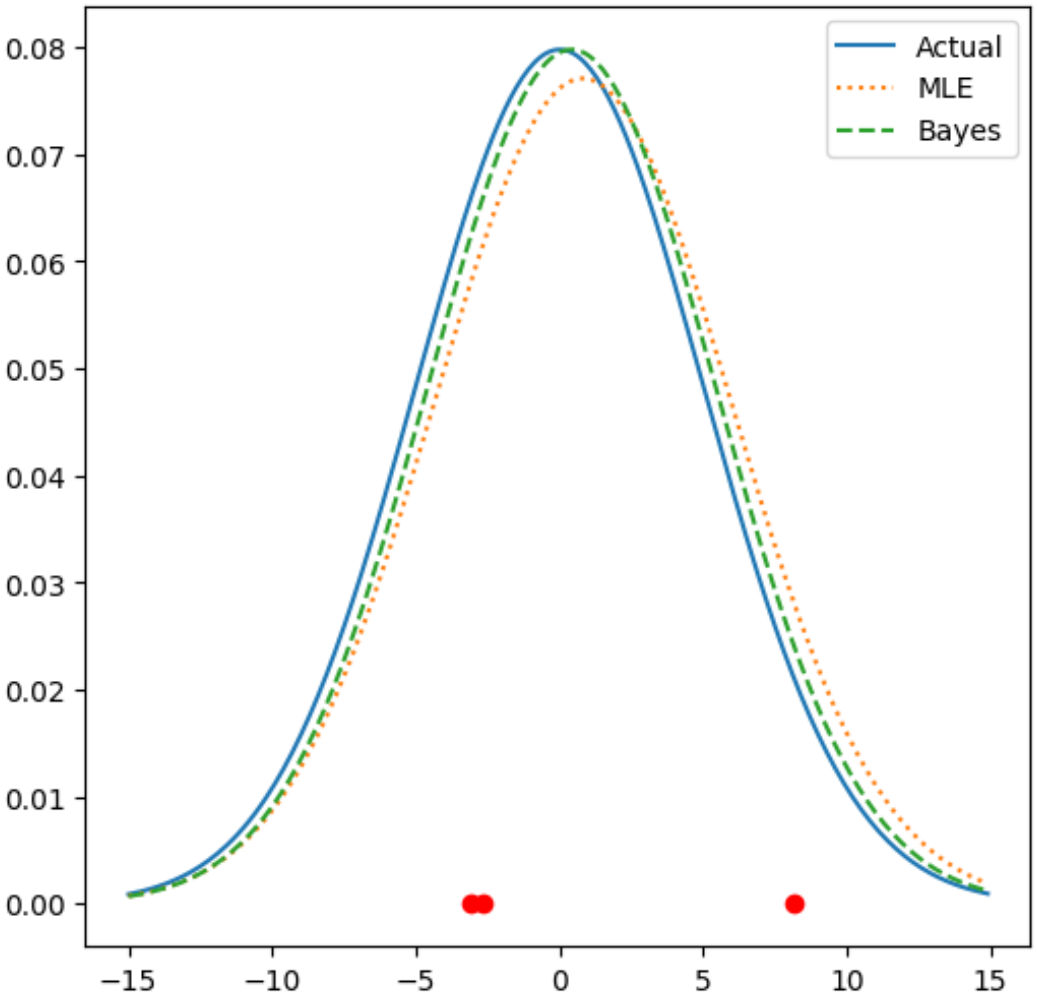
pt)

1. Plot the density function of $N(\mu, \sigma^2)$ with the x-axis ranging in `xrange` (an input argument)
2. Draw $N = 5$ samples from $N(\mu, \sigma^2)$ using `np.random.normal`. In the same figure from item 1, plot the N samples in by `plt.scatter`. Each sample x_i will lead to a red point at coordinate $(x_i, 0)$ (y-axis is 0).
3. Apply maximum-likelihood estimation (MLE) on these N samples to estimate the μ as μ_{MLE} and σ as σ_{MLE}. Then plot the density function of $N(\mu_{MLE}, \sigma^2_{MLE})$ in the same plot as item 1.
4. Call `bayes_estimator` you implemented before by using the N samples and a prior Gaussian with mean `priorMean` and standard deviation `priorStd` (both are input arguments). Denote the Bayes estimate of μ as μ_{Bayes}.
5. Plot the density function of $N(\mu_{Bayes}, \sigma^2)$ in the same figure as item 1. Note we do not consider a Bayes estimate of σ, and will just directly use the given σ for plotting. This is different from item 3 above.

Make sure that the three curves in the plot use different colors and line styles. At a good position, put the legend as "Actual", "MLE", and "Bayes".

```
def ex4_3(xrange, mu, sigma, priorMean, priorStd, N):
  '''
  Inputs:
    - xrange: a 1-D numpy array specifying the range of x to ma
    - mu: a real number as the mean of the Gaussian
    - sigma: a real number as the standard deviation of the Gau
    - N: an integer specifying how many samples to draw
  Outputs: nil
  '''
  np.random.seed(1)

  # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)***

  plt.figure(figsize = (6, 6))
  plt.plot(xrange, nrmf(xrange, mu, sigma), label='Actual', lir

  samples = np.random.normal(mu, sigma, N)
  plt.scatter(samples, np.zeros(samples.shape[0]), color = 'rec
```

```
  mu_mle = np.mean(samples)
  sigma_mle = np.sqrt(np.sum(np.square(samples - mu_mle)) / sam
  plt.plot(xrange, nrmf(xrange, mu_mle, sigma_mle), label='MLE'

  mu_bayes = bayes_estimator(samples, sigma, priorMean, priorSt
  plt.plot(xrange, nrmf(xrange, mu_bayes, sigma), label='Bayes'

  plt.legend()

  # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

'''
This is the real test you will be graded on:
'''
mu, sigma = (0, 5)
priorMean, priorStd = (mu, 3)
x_start, x_end, step= (mu - 3*sigma, mu + 3*sigma, 0.1)
xrange = np.arange(x_start, x_end, step)
N = 3
ex4_3(xrange, mu, sigma, priorMean, priorStd, N)
```



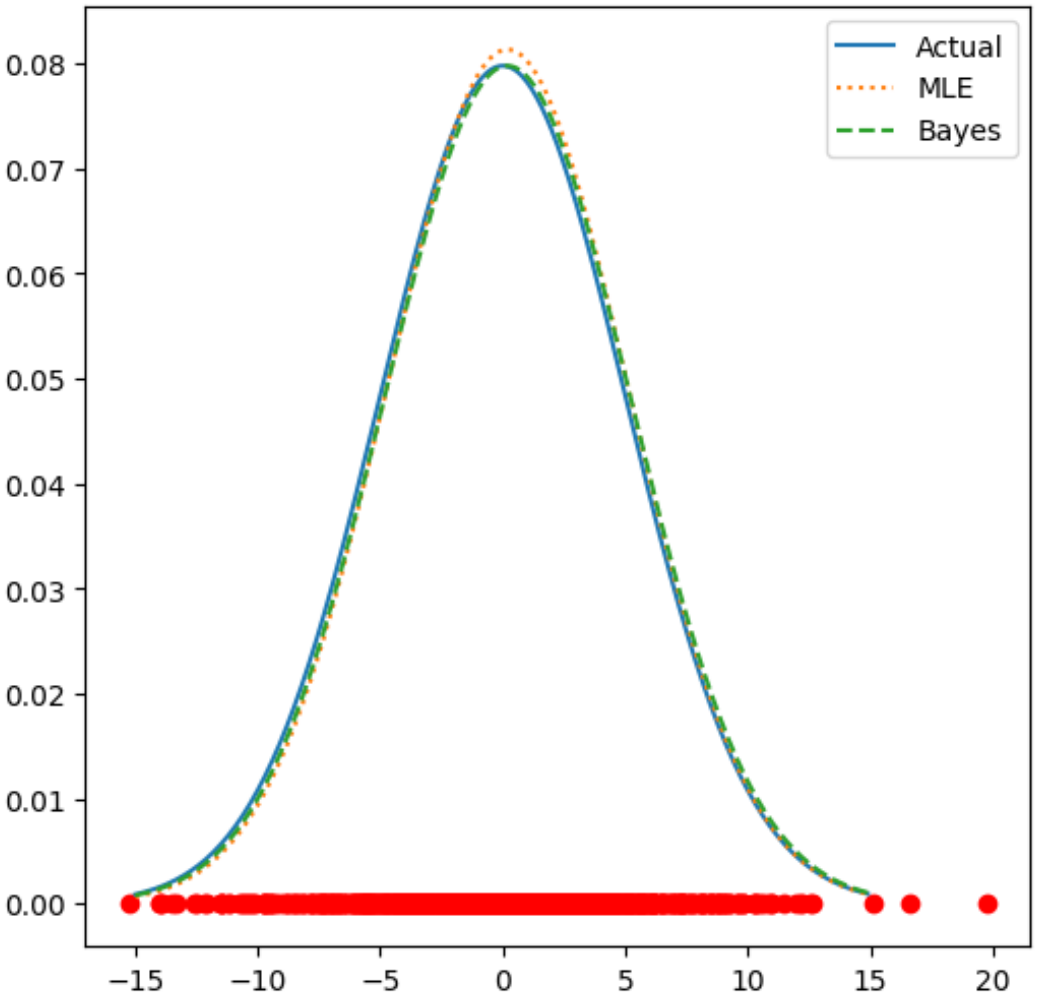
2.4 Call `ex4_3` with the same input arguments as before, but now set N to 1000 (5 pt)

Discuss briefly how the new plot changes from the previous one by writing in the paragraph below. Pay attention to μ, μ_{MLE}, and μ_{Bayes}.

With the increase in the number of samples the MLE is close to the original density function

```
'''
This is the real test you will be graded on:
'''
mu, sigma = (0, 5)
priorMean, priorStd = (mu, 3)
x_start, x_end, step= (mu - 3*sigma, mu + 3*sigma, 0.1)
xrange = np.arange(x_start, x_end, step)
N = 1000
```

```
ex4_3(xrange, mu, sigma, priorMean, priorStd, N)
```



Problem 3: Exercise 4.6 of Alpaydin (26 pt)

For a two-class problem, suppose the likelihood models for both classes are Gaussians, and their covariances are different. Visualize the posterior probability, and use parametric classification to estimate the discriminant points.

3.1 Compute the likelihood and posterior probability (8 pt)

Let $p(x|C_1) = N(\mu_1, \sigma_1^2)$ and $p(x|C_2) = N(\mu_2, \sigma_1^2)$. On the points

specified by `xrange`, compute likelihood $p(x|C_1)$ and posterior $p(C_1|x)$, along with $p(x|C_2)$ and $p(C_2|x)$.

Note, your code should not have any for loop because the function `nrmf` you implemented before can take an array as the first input argument.

```
def comp_posterior(mu1, sigma1, p1, mu2, sigma2, xrange):
  """
  Compute the likelihood and posterior proability for x values
  Inputs:
  - mu1: a real number specifying the mean of Gaussian distribu
  - sigma1: a real number specifying the standard deviation of
  - p1: a real number in [0, 1] specifying the prior probabilit
        P(C2) will be automatically inferred by 1 - p1.
  - mu2: a real number specifying the mean of Gaussian distribu
  - sigma2: a real number specifying the standard deviation of
  - xrange: a 1-D numpy array specifying the range of x to eval

  Outputs:
  - l1: a 1-D numpy array in the same size as xrange, recording
  - post1: a 1-D numpy array in the same size as xrange, record
  - l2: a 1-D numpy array in the same size as xrange, recording
  - post2: a 1-D numpy array in the same size as xrange, record
  """
  np.random.seed(1)

  # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)***
  # p(x|C1)
  l1 = nrmf(xrange, mu1, sigma1)

  # p(x|C2)
  l2 = nrmf(xrange, mu2, sigma2)

  # p(C1|x)
  post1 = (l1 * p1) / (p1 * l1 + (1 - p1) * l2)

  # p(C2|x)
  post2 = l2 * (1 - p1) / (p1 * l1 + (1 - p1) * l2)
  # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

  return l1, post1, l2, post2

"""
Unit test below.
```

```
Should print:
[0.00443185 0.05399097 0.24197072]
[0.9999999  0.87502269 0.10818288]
[2.97030006e-10 5.14092999e-03 1.32980760e+00]
[1.00532537e-07 1.24977307e-01 8.91817115e-01]
"""
mu1 = 3.0
sigma1 = 1.0
p1 = 0.4
mu2 = 2.0
sigma2 = 0.3
xrange = np.arange(0,3,1)

l1, post1, l2, post2 = comp_posterior(mu1, sigma1, p1, mu2, sig
print(l1)
print(post1)
print(l2)
print(post2)
```

```
[0.00443185 0.05399097 0.24197072]
[0.9999999  0.87502269 0.10818288]
[2.97030006e-10 5.14092999e-03 1.32980760e+00]
[1.00532537e-07 1.24977307e-01 8.91817115e-01]
```

3.2 Compute the two discriminant points (8 pt)

Use the results in Exercise 4.4 of Alpaydin's book to compute, analytically, the two values of x such that $p(C_1|x) = p(C_2|x)$. Denote them as x_1 and x_2.

```
def find_dpoint(mu1, sigma1, p1, mu2, sigma2):
  '''
  Find the discriminant points of two Gaussians
  Inputs:
  - mu1: a real number specifying the mean of Gaussian distribu
  - sigma1: a real number specifying the standard deviation of
  - p1: a real number in [0, 1] specifying the prior probabilit
        P(C2) will be automatically inferred by 1 - p1.
  - mu2: a real number specifying the mean of Gaussian distribu
  - sigma2: a real number specifying the standard deviation of
  Output:
  - x: a 1-D numpy array with two elements, recording the discr
  '''
```

```
    # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
    a = (1 / (2*sigma2**2)) - (1 / (2*sigma1**2))
    b = (mu1 / sigma1**2) - (mu2 / sigma2**2)
    c = (mu2**2/(2*sigma2**2) - mu1**2/(2*sigma1**2)) + np.log(si

    x1 = (-b + np.sqrt(b**2 - 4*a*c)) / (2*a)
    x2 = (-b - np.sqrt(b**2 - 4*a*c)) / (2*a)
    # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

    return np.array([x1, x2])

"""
Unit test below.
Should print:
[2.55457643 1.24762137]
"""
mu1 = 3.0
sigma1 = 1.0
p1 = 0.4
mu2 = 2.0
sigma2 = 0.3

dpoint = find_dpoint(mu1, sigma1, p1, mu2, sigma2)
print(dpoint)
```

```
[2.55457643 1.24762137]
```

3.3 Plot using the previous functions (10 pt)

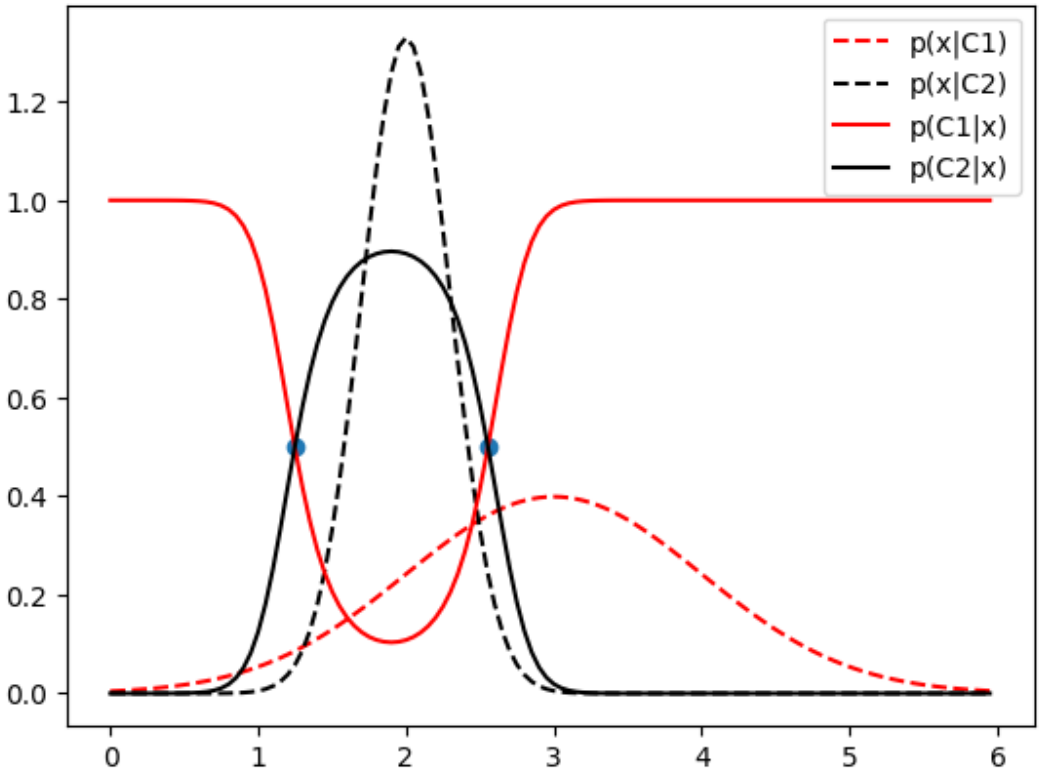
1. Plot the density of $p(x|C_1)$ and $p(x|C_2)$ in one plot, with the x-axis range being the input argument `xrange`. Use **dashed** line style ('–'), and color $p(x|C_1)$ as red, and $p(x|C_2)$ as black.

2. In the previous figure, plot two more curves $p(C_1|x)$ and $p(C_2|x)$, both as a function of x as given by `xrange`. Use **solid** line style ('-'), and color $p(C_1|x)$ as red, and $p(C_2|x)$ as black.

3. Use `find_dpoint` to find the two discriminant points x_1 and x_2. Put a small solid blue circle at $(x_1, p(C_1|x_1))$ and $(x_2, p(C_1|x_2))$. Hint: you may want to reuse `comp_posterior` here.

Note that 1 and 2 can be accomplished by using the function `comp_posterior`.

```
def ex4_6(mu1, sigma1, p1, mu2, sigma2, xrange):
  '''
  Plot the likelihood and posterior probabilities for two class
    and plot the discrminant points.
  Inputs:
  - mu1: a real number specifying the mean of Gaussian distribu
  - sigma1: a real number specifying the standard deviation of
  - p1: a real number in [0, 1] specifying the prior probabilit
        P(C2) will be automatically inferred by 1 - p1.
  - mu2: a real number specifying the mean of Gaussian distribu
  - sigma2: a real number specifying the standard deviation of
  - xrange: a 1-D numpy array specifying the range of x to eval
  Outputs:
    Nil
  '''

  # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)***
  l1, post1, l2, post2 = comp_posterior(mu1, sigma1, p1, mu2, s


  plt.plot(xrange, l1, color='red', linestyle='dashed', label='
  plt.plot(xrange, l2, color='black', linestyle='dashed', label


  plt.plot(xrange, post1, color='red', linestyle='solid', label
  plt.plot(xrange, post2, color='black', linestyle='solid', lab

  dpoint = find_dpoint(mu1, sigma1, p1, mu2, sigma2)
  a, post_1, b, post_2 = comp_posterior(mu1, sigma1, p1, mu2, s
  plt.scatter(dpoint, post_1)
  plt.legend()

  # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

mu1 = 3.0
sigma1 = 1.0
p1 = 0.4
mu2 = 2.0
sigma2 = 0.3
xrange = np.arange(0,6,0.05)

ex4_6(mu1, sigma1, p1, mu2, sigma2,  xrange)
```



Problem 4: Exercise 4.7 of Alpaydin (0 pts)

Assume a linear model and then add 0-mean Gaussian noise to generate a sample. Divide your sample into two as training and validation sets. Use linear regression using the training half. Compute error on the validation set. Do the same for polynomials of degrees 2 and 3 as well. You may find **np.polyfit()** is useful.

The code has been fully proivded and you are expected to digest it fully. Also play with the code to try anything you like. There is nothing to submit for this question.

```
def ex4_7(nSample, n_train, degree):
  """
  Inputs:
  - degree: List, degree list of polynoimial function.
  - nSample: Integer, number of samples you want to draw
  - n_train: Integer, number of samples in the training set
```

```
  Output:
  - error: List, error of different degrees on validation set
  """

  #generate samples
  np.random.seed(1)
  # X = np.arange(nSample)
  X = np.random.rand(nSample)
  Y = X + np.random.standard_normal(nSample)

  #split samples into training set and validation set
  order = np.random.permutation(len(X))
  X_train = X[order[:n_train]]
  Y_train = Y[order[:n_train]]
  X_test = X[order[n_train:]]
  Y_test = Y[order[n_train:]]

  error = np.zeros_like(degree)

  for d in degree:
  #train fit
    A = np.polyfit(X_train, Y_train, d)
    Y_predict = np.zeros_like(Y_test)

    #compute the error
    for i in range(len(A)):
      Y_predict += A[i]*np.power(X_test, d-i)

    error = np.power((Y_predict-Y_test),2)
    error[d] = (1.0/(2*nSample)) * error.sum(axis=0)
    print ('The error of polynoimial degree {degree} : {error}'
  return error

nSample = 100
n_train = 50
degree = [1, 2, 3]
error = ex4_7(nSample, n_train, degree)
```

```
The error of polynoimial degree 1 : 0.1572958856196469
The error of polynoimial degree 2 : 0.1736232341120441
The error of polynoimial degree 3 : 0.17600437231468824
```

Problem 5: Exercise 5.2 of Alpaydin (32

pt)

Generate a sample from a 2-dimensional normal density $N(\mu, \Sigma)$, calculate m and S, and compare them with μ and Σ. Check how your estimates change as the sample size changes.

Specificially, draw $N = 10, 50, 500$ samples. For each value of N, generate a scatter plot of the samples drawn. On each plot, also include a contour of the original density $N(\mu, \Sigma)$.

5.1 Implement the function that computes the density of a multi-variate Gaussian distribution (9 pt)

```
def mvar(x, mu, Sigma):
  '''
    Computes the density function of a multi-variate Gaussian d
      with mean mu and covariance matrix Sigma, evaluated at x.
    This is the multi-variate version of nrmf.
    Different from nrmf, x encodes a single location only.

    Inputs:
    - x: 1-D numpy array specifying the single position where t
    - mu: 1-D numpy array specifying the mean of 2-dimensional
    - Sigma: 2-D numpy array specifying the covariance of 2-dim
    Outputs:
    - p: a real number specifying the probability density of x
  '''

  # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)***

  d = Sigma.shape[0]
  p = (1 / ((2*np.pi)**(d / 2) * np.linalg.det(Sigma)**(1/2)))

  # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

  return p

"""
Unit test:
should print: p = 0.011828135195080136
"""
```

```
Sigma = np.array([[1, -1],
                  [-1, 5]])

mu = np.array([1.5, 2])
x = np.array([3, 3])
p = mvar(x, mu, Sigma)
print(p)
```

```
0.011828135195080136
```

5.2 Create the meshgrid to plot the contour for the density function (9 pt)

Given a range of x and y values (both as a vector), create a mesh grid as their cross product. Most parts of the function have been done for you, and you will only need to implement one line of code that evaluates the Gaussian density using the `mvar` function. Read the code carefully and digest the `meshgrid` function. It will be very important for plotting 2-variable functions.

You may look at the code in Section 5.4 below and see how to make `comp_density_grid` compatible with its application there. As such, we are not providing a test case here, but you should make sure that the results in Section 5.4 look reasonable.

```
def comp_density_grid(x, y, mu, Sigma):
  '''
  Generate the meshgrid of plotting the 2-variable Gaussian der
  Inputs:
    - x: 1-D numpy array specifying the ticks of x-axis
    - y: 1-D numpy array specifying the ticks of y-axis
    - mu: 1-D numpy array specifying the mean of 2-dimensional
    - Sigma: 2-D numpy array specifying the covariance of 2-dim
  Outputs:
    - X: shaped x by using np.meshgrid (len(y), len(x))
    - Y: shaped y by using np.meshgrid (len(y), len(x))
    - f: 2-D numpy array specifying the probability density of
  '''
  X, Y = np.meshgrid(x, y)  # Both X and Y are shaped (len(y),
  len_x = np.size(x)
  len_y = np.size(y)
  f = np.empty([len_y, len_x])
```

```
  for i in range(len_x):
    for j in range(len_y):
      # Next we need to assign the values for the f matrix
      # It is quite tricky. Check out
      # Manual: https://numpy.org/doc/stable/reference/generate
      # Pay attention to the 'notes' section in the manual page
      # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE

      f[j][i] = mvar([x[i], y[j]], mu, Sigma)

      # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*
  return X, Y, f
```

5.3 Draw samples from a multivariate Gaussian and re-estimate its parameters (9 pt)

To draw a sample from an n-dimensional Gaussian $N(\mu, \Sigma)$, one needs to take the following steps:

1. Perform a Cholesky decomposition on Σ, i.e., find an n-by-n upper-triangular matrix R such that $R^T R = \Sigma$. Since Σ is positive semi-definite, there must be such a real-valued matrix R.

2. Draw n number of iid samples from $N(0, 1)$ using `np.random.standard_normal`. Stack them together into an n-dimensional vecotr x.

3. Output a sample computed by $R^\top x + \mu$.

The function `sample_multivariate_normal` should first loop over steps 1-3 to draw N samples, and then use these samples to estimate the mean and covariance matrix of the Gaussian.

Now implement this procedure. You code should not import any module and can directly call `scipy.linalg.cholesky`.

```
def sample_multivariate_normal(mu, Sigma, N):
  '''
  Draw samples from a multivariate Gaussian and then re-estimat
  Inputs:
    - mu: 1-D numpy array specifying the mean of 2-dimensional
    - Sigma: 2-D numpy array specifying the covariance of 2-dim
```

```
    - N: Integer, the number of samples (num_sample)
    ...
  Outputs:
    - sample: 2-D numpy array of drawn samples from a multivari
    - mean: 1-D numpy array specifying the mean of drawn sample
    - cov: 2-D numpy array specifying the covariance of drawn s
  '''
  np.random.seed(1)
  # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)***

  R = scipy.linalg.cholesky(Sigma)
  sample = np.zeros((N, 2))
  for i in range(N):
    x = np.random.standard_normal(size = 2)
    sample[i] = R.T @ x + mu
  mean = np.mean(sample, axis=0)
  cov = np.cov(sample.T)

  # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

  return sample, mean, cov

  '''
  Unit test:
  It is slightly difficult to test the implementation because t
  Depending on your implementation, it is hard to specify a "co
  So one way to check is to draw a large number of samples (e.g
  then estimate the mean and covariance from it.
  Finally, compare them with the input arguments mu and Sigma.
  You can implement this test yourself, and the auto-grader wil
  '''
```

5.4 Generate contours of Gaussian densities and samples (5 pt)

1. Generate contours of Gaussian densities,
2. Draw N samples from it,
3. Estimate the mean and covariance based on the N samples,
4. Generate the contour of the estimated Gaussian.

We will vary $N = 10, 50, 500$. The code has been done for you, and you should carefully and thoroughly comprehend it. In particular, see how the `contour` function is used. Your implementation in the previous steps

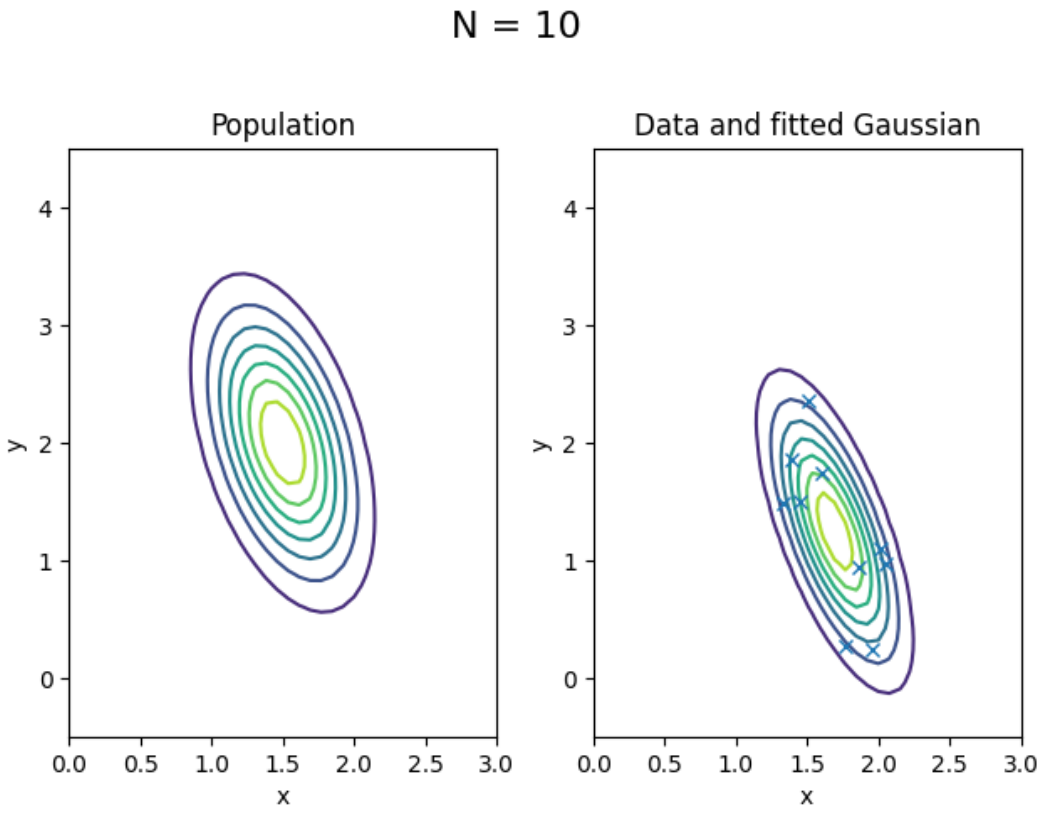
should make the plots here reasonable. There is no code to submit for this step.

Discuss briefly what observations you can make from the plots.

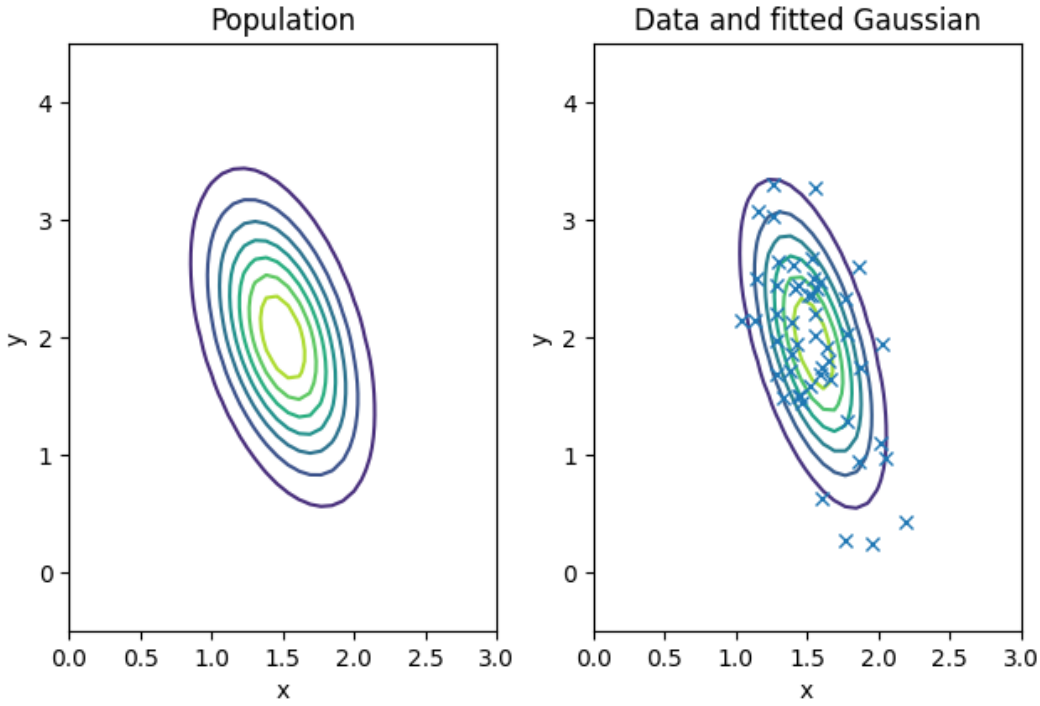
My observation is : When the N increases, the Data and fitted Gaussian becomes similar to the actual one

```
def ex5_2(Nlist):
  np.random.seed(1)
  Sigma = np.array([[0.1, -0.1],
            [-0.1, 0.5]])

  m = np.array([1.5, 2])
  x = np.linspace(0, 3, 40)
  y = np.linspace(-0.5, 4, 50)

  for N in Nlist:

    # Generate contours of Gaussian densities
    # This subplot is the same for different values of N.
    X, Y, f = comp_density_grid(x, y, m, Sigma)
    fig = plt.figure()
    fig.suptitle(f"N = {N}", va='bottom', fontsize=16)
    ax1 = fig.add_subplot(121)    # Digest the syntax here
    ax1.title.set_text('Population')
    ax1.contour(X, Y, f)
    ax1.set_xlabel('x')
    ax1.set_ylabel('y')
    ax1.set_ylim(-0.5,4.5)

    ax2 = fig.add_subplot(122)
    ax2.title.set_text('Data and fitted Gaussian')
    # Draw N samples from the Gaussian,
    # and estimate the mean and covariance based on these sampl
    dta, mean, cov = sample_multivariate_normal(m, Sigma, N)

    # Generate the contour of the estimated Gaussian.
    X, Y, f = comp_density_grid(x, y, mean, cov)
    ax2.contour(X, Y, f)

    ax2.plot(dta[:,0], dta[:,1], 'x')
    ax2.set_xlabel('x')
    ax2.set_ylabel('y')
```

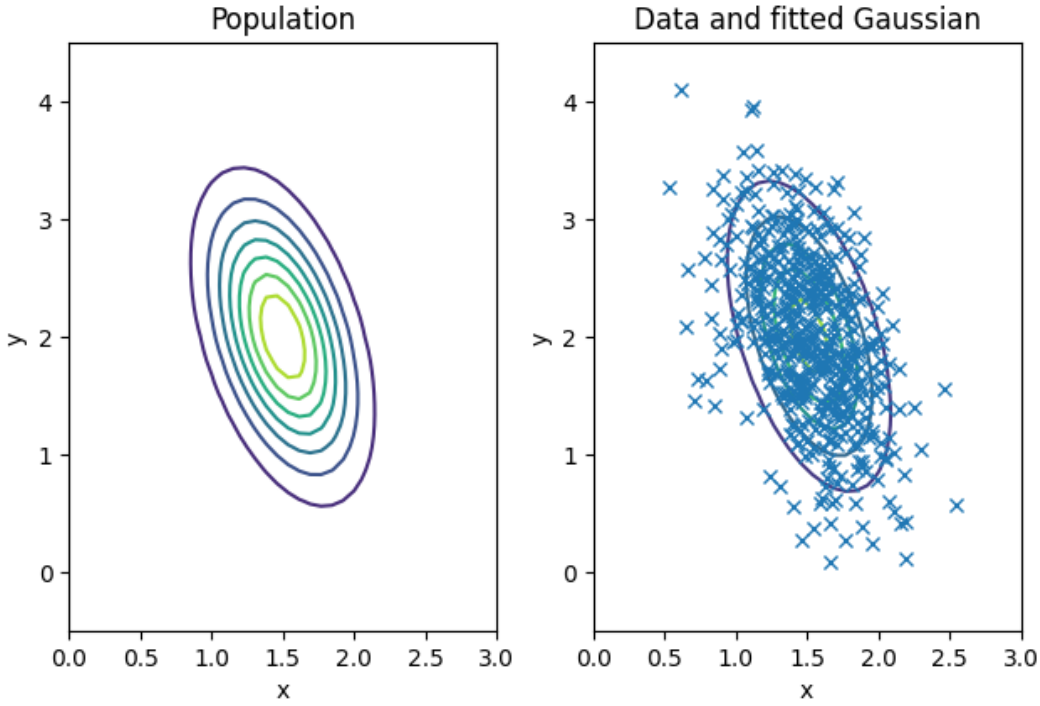
```
    ax2.set_ylim(-0.5,4.5)
    plt.tight_layout()

N = [10, 50, 500]
ex5_2(N)
```



N = 50


Population
Data and fitted Gaussian
y
x
0
1
2
3
4
0.0
0.5
1.0
1.5
2.0
2.5
3.0

N = 500


Population
Data and fitted Gaussian
y
x
0
1
2
3
4
0.0
0.5
1.0
1.5
2.0
2.5
3.0

Problem 6: Exercise 5.6 of Alpaydin (0 pt)

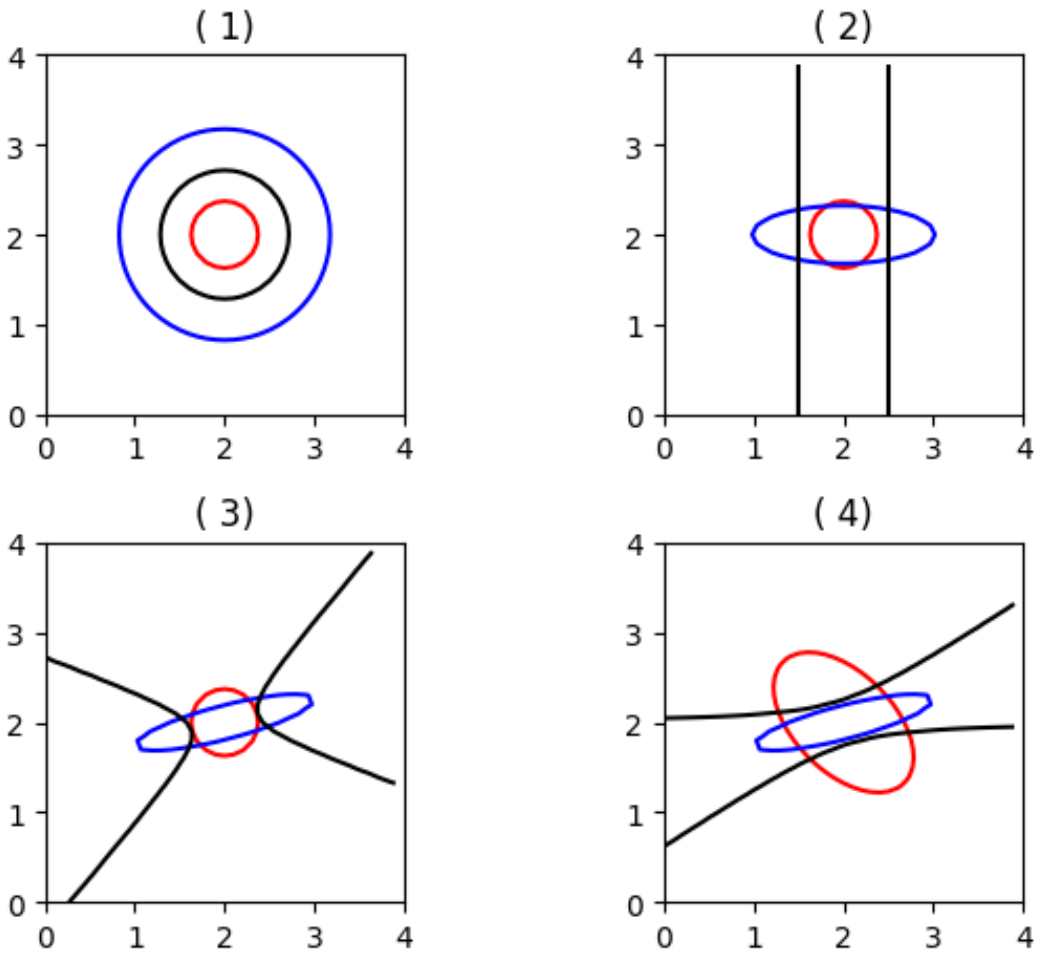
Let us say in two dimensions, we have two classes with exactly the same mean. What type of boundaries can be defined?

Digest the code below and appreciate the four cases. We plot in black the decision boundary where $p_1 - p_2 = 0$. We assume $p(C_1) = p(C_2) = 0.5$. We generate the contour plot of the likelihood $p(x|C_1)$ and $p(x|C_2)$ at the level of 1, in red and blue respectively. There is no code to submit for this question.

```
def cal_likli_pos(x, y, m1, m2, sigma1, sigma2):
  '''
  Generate the meshgrid of plotting
      1) two Gaussian density functions of two variables (outpu
      2) the posterior probability of the two classes (p1, p2)
  Inputs:
    - x, y: 1-D numpy arrays to specify the x and y coordinates
    - m1, m2, sigma1, sigma2: the mean and covariance of the tw
  '''
  X, Y = np.meshgrid(y, x)
  len_x = np.size(x)
  len_y = np.size(y)

  f1 = np.zeros((len_y, len_x))
  f2 = np.zeros((len_y, len_x))
  p1 = np.zeros((len_y, len_x))
  p2 = np.zeros((len_y, len_x))
  for i in range(len_y):
    for j in range(len_x):
        f1[i][j] = mvar(np.array([x[j], y[i]]), m1, sigma1)
        f2[i][j] = mvar(np.array([x[j], y[i]]), m2, sigma2)
        p1[i][j] = f1[i][j]/(f1[i][j]+f2[i][j])
        p2[i][j] = 1 - p1[i][j]

  return f1, f2, p1, p2
```

```
def ex5_6(mean, sg1, sg2):
  if (sg1.shape != sg2.shape):
    return 0
  x = np.arange(0, 4, 0.1)
  y = np.arange(0, 4, 0.1)
```

```
  n = sg1.shape[0]
  for i in range(n):
    f1, f2, p1, p2 = cal_likli_pos(x, y, mean, mean, sg1[i], sg
    plt.subplot(int(np.sqrt(n)),int(np.sqrt(n)), i+1)
    plt.title('({0:2d})'.format(i+1))
    plt.contour(x, y, f1, 1, colors='r')
    plt.contour(x, y, f2, 1, colors='b')
    plt.contour(x, y, p1-p2, 0, colors='k')   # plot the bounda
    plt.xticks(np.arange(0, 5, step=1))
    plt.yticks(np.arange(0, 5, step=1))
    plt.gca().set_aspect('equal')

  plt.tight_layout()

mean = np.array([2.0,2.0])
sg1 = np.array([[[0.1,0],[0,0.1]],[[0.1,0],[0,0.1]],[[0.1,0],[0
sg2 = np.array([[[1,0],[0,1]], [[1,0],[0,0.1]], [[1,0.25],[0.25
ex5_6(mean, sg1, sg2)
```



Problem 7: Based on Exercise 5.3 of Alpaydin (15 pt)

Generate samples from two multivariate normal densities $N(\mu_i, \Sigma_i)$ ($i = 1, 2$), and use these samples to calculate the Bayes' optimal discriminant for the four cases in Table 5.1. For simplicity, assume $N(\mu_i, \Sigma_i)$ is a 2-dimensional Gaussian density for both $i = 1$ and 2.

We can consider $N(\mu_i, \Sigma_i)$ as the likelihood $p(x|C_i)$. We will assume a uniform prior $p(C_1) = p(C_2) = 0.5$. Our overall experiment is set up as follows.

1. Generate $N = 5$ samples from the given mean $N(\mu_i, \Sigma_i)$.
2. Estimate μ_i from the samples and denote the estimates as $mean_i$.
3. For k = 4, 3, 2, 1 **(the construction of covariance estimate in the four cases will be covered by Section 7.1)**
    - If k == 4, estimate the covariances Σ_i as "Different, Hyperellipsoidal". Denote the result as cov_i. Refer to the last row of Table 5.1 of Alpaydin.
    - If k == 3, estimate the covariances Σ_i as "Shared, Hyperellipsoidal". This shared covariance matrix can be computed as $(cov_1 + cov_2)/2$ based on the result of k == 4.
    - If k == 2, estimate the covariances Σ_i as "Shared, Axis-aligned". Denote the shared covariance matrix from $k = 3$ as S. Then the new shared covariance matrix is just a diagonal matrix whose (i, i)-th entry is S_{ii}. In other words, just set the off-diagonal entries of S to 0.
    - If k == 1, estimate the covariances Σ_i as "Shared, Hyperspheric". Denote the shared covariance matrix from $k = 2$ as S. Then the new shared covariance matrix is just $a \cdot I$, where I is the identity matrix and a is the mean of the diagonal entries of S.
4. For k = 4, 3, 2, 1
    - Plot the contour of the density of the two Gaussians with the

estimated mean ($mean_i$) and covariance (cov_i for $k = 4$, etc).

- Plot the optimal discriminant (i.e., decision boundary) for the two classes, i.e., where $p(C_1|x) = p(C_2|x) = 0.5$. You can use the same technique or code from Problem 6.

7.1 Compute the covariance estimation for the four different assumptions (10 pt)

We first implement the covariance matrix for the four cases in the function `comp_CovList`. It takes cov_i from the standard estimation of the two Gaussians assuming they are different and hyperellipsoidal (see the case $k = 4$ above); you can see how they are computed from the code in Section 7.2. The output is two lists: `cov1List` and `cov2List`. `cov1List` is a list of four entries, where `cov1List[k-1]` is the estimate of Σ_1 in the above case k. Similarly for `cov2List`.

```
def comp_CovList(cov1, cov2):
  '''
  Implement the covariance matrix for the four cases
  Inputs:
  - cov1: the estimate of covariance matrix for the first Gauss
  - cov2: the estimate of covariance matrix for the second Gaus
        Note cov1 and cov2 are estimated separately, just like
  Outputs:
    - cov1List: a list of four entries, where cov1List[k-1] is
    - cov2List: a list of four entries, where cov2List[k-1] is
  '''
  cov1List = list()
  cov2List = list()

  for k in range(4):
    # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*

    cov_avg = (cov1 + cov2)/2

    if k == 3:
      cov1List.append(cov1)
      cov2List.append(cov2)

    if k == 2:
      cov1List.append(cov_avg)
      cov2List.append(cov_avg)
```

```
    if k == 1:
      cov_diagonal = np.zeros((cov_avg.shape[0], cov_avg.shape
      np.fill_diagonal(cov_diagonal, np.diag(cov_avg))
      cov1List.append(cov_diagonal)
      cov2List.append(cov_diagonal)


    if k == 0:
      diagonal_element = np.mean(np.diag(cov_avg))
      cov_diagonal_same_element = diagonal_element * np.identit
      cov1List.append(cov_diagonal_same_element)
      cov2List.append(cov_diagonal_same_element)

    # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)***
  return cov1List, cov2List
```

7.2 Use `compCovList` to plot Gaussian likelihood and optimal discriminant. (5 pt)

The code has been provided. You should read and digest the code. There is no code to submit.

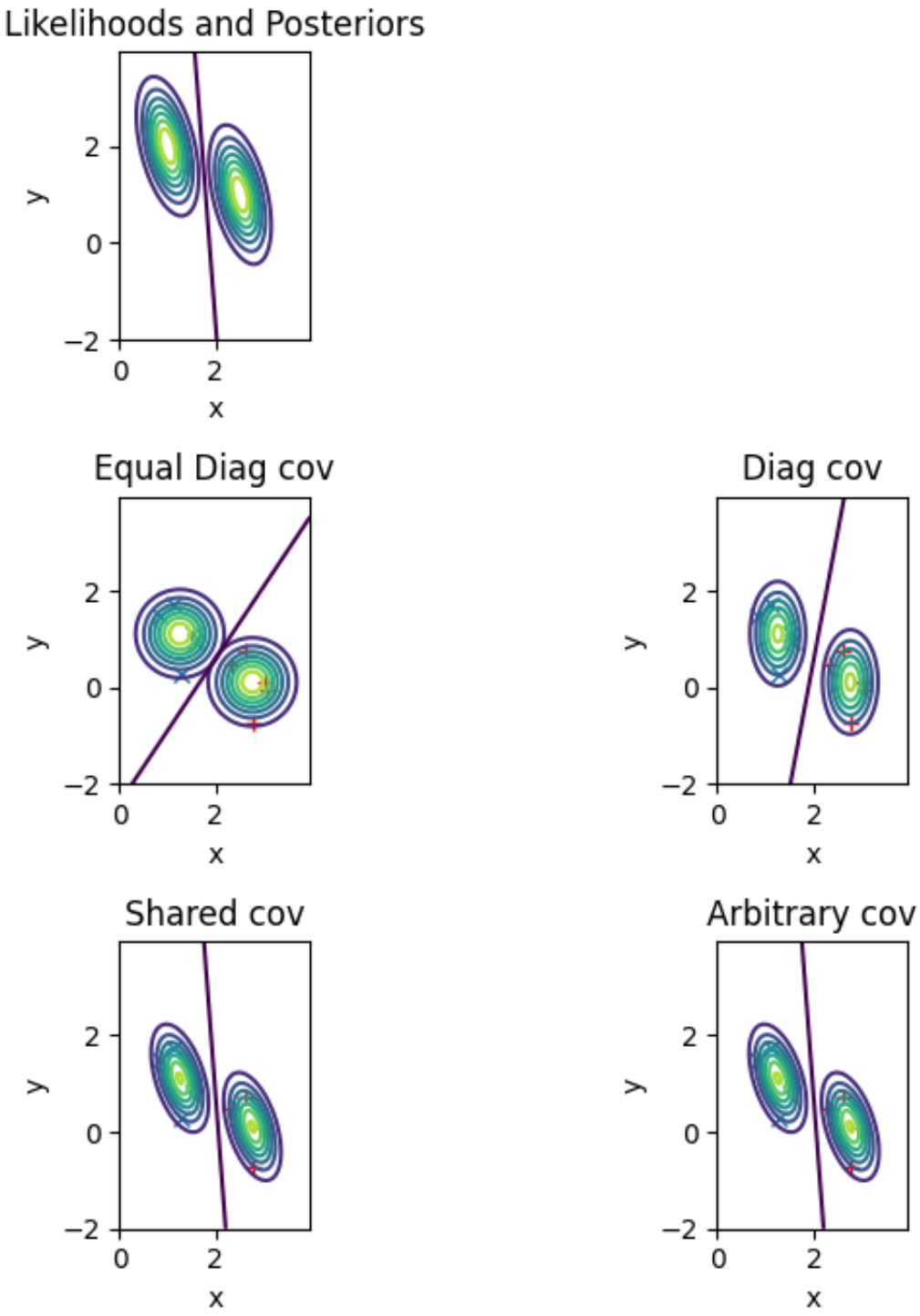
Now consider which one of the four assumptions in Table 5.1 of Alpaydin is the best for our samples from the two Gaussians. If it is not "Different, Hyperellipsoidal" ($k = 4$), explain why we cannot benefit from such a high capacity model. If it is not "Shared, Hyperspheric" ($k = 1$), explain why it does not work well.

My observation is: When k=4, we get a very small number of samples N=5 for each contour, so it is not the best model. When k=1, we use a single value(mean) for all the diagonal elements of the covariance Matrix, so it does not represent the correct covariance so it is not the best model too. The best model is when k=3 because the we use a shared covariance Matrix but all the elements are neither shared nor it is a diagonal Matrix

```
def ex5_3():
  np.random.seed(1)

  mu1 = np.array([1, 2])
  sigma1 = np.array([[0.1, -0.1],
          [-0.1, 0.5]])
```

```
mu2 = np.array([2.5, 1])
sigma2 = sigma1

min_x, max_x = (0, 4)
min_y, max_y = (-2, 4)

#xrange and yrange are the range for plotting
xrange = np.arange(min_x, max_x, step=1)
yrange = np.arange(min_y, max_y, step=1)
x = np.arange(min_x, max_x, 0.05)
y = np.arange(min_y, max_y, 0.05)

f1, f2, p1, p2 = cal_likli_pos(x, y, mu1, mu2, sigma1, sigma2
plt.figure(figsize=(7,7))
plt.subplot(3,2,1)
plt.contour(x, y, f1)
plt.contour(x, y, f2)
plt.xlabel('x')
plt.ylabel('y')
plt.title('Likelihoods and Posteriors')
plt.contour(x, y, p1-p2, levels=[0])
plt.gca().set_aspect('equal')

#Sample from populations
N = 5
dta1, mean1, cov1 = sample_multivariate_normal(mu1, sigma1, N
dta2, mean2, cov2 = sample_multivariate_normal(mu2, sigma2, N

cov1List, cov2List = comp_CovList(cov1, cov2)

for k in range(4):
  if k == 3:
    plt.subplot(3,2,6)
    title = 'Arbitrary cov'
  elif k == 2:
    plt.subplot(3,2,5)
    title = 'Shared cov'
  elif k == 1:
    plt.subplot(3,2,4)
    title = 'Diag cov'
  else:
    plt.subplot(3,2,3)
    title = 'Equal Diag cov'

  plt.xlabel('x')
```

```
    plt.ylabel('y')

    # Plot the samples dta1 using 'x'
    plt.plot(dta1[:,0], dta1[:,1], 'x')
    # Plot the samples dta1 using '+r'
    plt.plot(dta2[:,0], dta2[:,1], '+r')

    f1, f2, p1, p2 = cal_likli_pos(x, y, mean1, mean2, cov1List
    # Plot the contour of f1 and f2
    plt.contour(x, y, f1)
    plt.contour(x, y, f2)
    # Plot the decision boundary
    plt.contour(x, y, p1-p2, levels=[0])

    plt.gca().set_aspect('equal')
    plt.title(title)

  plt.tight_layout()

ex5_3()
```



Submission Instruction

You're almost done! Take the following steps to finally submit your work.

1. After executing all commands and completing this notebook, save

your `Lab_4.ipynb` as a PDF file, named as `X_Y_UIN.pdf`, where `X` is your first name, `Y` is your last name, and `UIN` is your UIN. Make sure that your PDF file includes all parts of your solution, including the plots.

> - Print out all unit test case results before printing the notebook into a PDF.
> - If you use Colab, open this notebook in Chrome. Then File -> Print -> set Destination to "Save as PDF". If the web page freezes when printing, close Chrome and reopen the page. If Chrome doesn't work, try Firefox.
> - If you are working on your own computer, we recommend using the browser (not jupyter) for saving the PDF. For Chrome on a Mac, this is under *File->Print...->Open PDF in Preview*. When the PDF opens in Preview, you can use *Save...* to save it.
> - Sometimes, a figure that appears near the end of a page can get cut. In this case, try to add some new lines in the preceding code block so that the figure is pushed to the beginning of the next page. Or insert some text blocks.

2. Upload `X_Y_UIN.pdf` to Gradescope under `Lab_4_Written`.

3. A template of `Lab_4.py` has been provided. For all functions in `Lab_4.py`, copy the corresponding code snippets you have written into it, excluding the plot code. **Do NOT** copy any code of plotting figures and do not import **matplotlib**. This is because the auto-grader cannot work with plotting. **Do NOT** change the function names.

4. Zip `Lab_4.py` and `Lab_4.ipynb` (**2 files**) into a zip file named `X_Y_UIN.zip`. Suppose the two files are in the folder `Lab_4`. Then zip up the **two files inside the `Lab_4` folder**. **Do NOT zip up the folder `Lab_4`** because the auto-grader cannot search inside a folder. Submit this zip file to Gradescope under `Lab_4_Code`.

5. The autograder on Gradscope will be open all the time. We designed some simple test cases to help you check wehther your functions are executable. You will see the results of running autograder once you submit your code. Please follow the error messages to debug. Since those simple test cases are designed for debugging, it does not

guaranttee your solution will work well on the real dataset. It is your responsibility to make your code logically correct. Since all functions are tested in batch, the autograder might take a few minutes to run after submission.

If you *only* try to get real-time feedback from auto-grader, it will be fine to just upload `Lab_4.py` to `Lab_4_Code`. However, the final submission for grading should still follow the above point 4.

You can submit to Gradescope as often as you like. We will only consider your last submission before the deadline.